The Roots of Software Engineering*

Michael S. Mahoney
Princeton University

At the International Conference on the History of Computing held in Los Alamos in 1976, R.W. Hamming placed his proposed agenda in the title of his paper: "We Would Know What They Thought When They Did It."[1] He pleaded for a history of computing that pursued the contextual development of ideas, rather than merely listing names, dates, and places of "firsts". Moreover, he exhorted historians to go beyond the documents to "informed speculation" about the results of undocumented practice. What people actually did and what they thought they were doing may well not be accurately reflected in what they wrote and what they said they were thinking. His own experience had taught him that.

Historians of science recognize in Hamming's point what they learned from Thomas Kuhn's *Structure of Scientific Revolutions* some time ago, namely that the practice of science and the literature of science do not necessarily coincide. Paradigms (or, if you prefer with Kuhn, disciplinary matrices) direct not so much what scientists say as what they do. Hence, to determine the paradigms of past science historians must watch scientists at work practicing their science. We have to reconstruct what they thought from the evidence of what they did, and that work of reconstruction in the history of science has often involved a certain amount of speculation informed by historians' own experience of science. That is all the more the case in the history of technology, where up to the present century the inventor and engineer have as Derek Price once put it "thought with their fingertips", leaving the record of their thinking in the artefacts they have designed rather than in texts they have written.

Yet, on two counts, Hamming's point has special force for the history of computing. First, whatever the theoretical content of the subject, the main object of computing has been to do something, or rather to make the computer do something. Successful practice has been the prime measure of effective theory. Second, the computer embodies a historically unique relation of thinking and doing. It is the first machine for doing thinking. In the years following its creation and its introduction into the worlds of science, industry, and

* An expanded version of a lecture presented at the CWI on 1 February 1990. It is based on research generously supported by the Alfred P. Sloan Foundation.

1. Published in N. Metropolis, J. Howlett, G.-C. Rota (eds.), *A History of Computing in the Twentieth Century: A Collection of Essays* (N.Y.: Academic Press, 1980), 3-9.

business, both the device and the activities involved in its use were new.

It is tempting to say they were unprecedented, were that not to beg the question at hand. Precedents are what people find in their past experience to guide their present action. Conversely, actions usually reflect the guidance of experience. Nothing is really unprecedented. Faced with a new situation, people liken it to familiar ones and shape their response on the basis of the perceived similarities. In the case of the computer, what was new was the reliable electronic circuitry that made its underlying theoretical structure realizable in practice. At heart, it was a Turing Machine that operated within the constraints of real time and space. That much was unprecedented. Beyond that, precedent shaped the computer. The Turing Machine was an open schema for a potentially infinite range of particular applications. How the computer was going to be used depended on the experience and expectations of the people who were going to use it or were going to design it for others to use.

As part of a history of the development of the computer industry from 1950 to 1970 focusing on the origins of the "software crisis", I am currently trying to determine what people had in mind when they first began to talk about "software engineering". Although one writer has suggested that the term originated in 1965,[2] it first came into common currency in 1967 when the Study Group on Computer Science of the NATO Science Committee called for an international conference on the subject. As Brian Randell and Peter Naur point out in the introduction to their edition of the proceedings, "The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be [based] on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."[3]

It is not entirely clear just what the Study Group meant to provoke, since that statement opens several areas of potential disagreement. Just what are the "types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering"? What would their counterparts look like for software engineering? What role does engineering play in manufacture? Could one assign such a role to software engineering? Can software be manufactured? Clearly, the Study Group thought the answer to the last question was yes, but it offered no definitive answers to the others, and the proceedings of the conference, along with the literature since, reveal a range of interpretations among the practitioners who were to become software engineers.

2. Brian Randell ("Software Engineering in 1968", *Proc. 4th Intern. Conf. on Software Engineering* [Munich, 1979], 1) cites R.M. Gordon's attribution of the term to J.P. Eckert at the Fall Joint Computer Conference in 1965, but the transcript of the one panel discussion in which Eckert participated shows no evidence of the term "software engineering". D.T. Ross claims the term was used in courses he was teaching at MIT in the late '50s; cf. "Interview: Douglas Ross Talks About Structured Analysis", *Computer* (July 1985), 80-88.

3. Peter Naur, Brian Randell, J.N. Buxton (eds.), *Software Engineering: Concepts and Techniques* (NY: Petrocelli/Charter, 1976; hereafter *NRB*).

Their differences extended beyond the realm of software to the nature of engineering itself. What some viewed as applied science, others took to be a body of techniques of design, while still others thought in terms of organization and management. Each point of view encompassed its own models and touchstones, in most cases implicitly and perhaps even unconsciously. Small wonder that conferences and symposia on software engineering through the '70s and into the '80s regularly began with keynotes addressed to defining the subject and to answering the question: Are we there yet? It depended on where one thought "there" was.

If one could not define "software engineering" at the time, neither could one point to its practice. The term was not coined to characterize an ongoing activity but rather to express a desire for one. By 1967, when the computer industry was less than twenty years old, people felt the need for software engineering, even if they were not sure what it was. A brief look at the origins of computing may help to explain such apparently strange behavior.

The electronic digital stored-program computer marks the convergence of two essentially independent lines of development tracing back to the early nineteenth-century, namely the design of mechanical calculators capable of automatic operation and the development of mathematical logic. In outline, at least, those stories are reasonably well known and need no repetition here.[4] Viewing them as convergent rather than coincident emphasizes that the computer emerged as the joint product of electrical engineering and theoretical mathematics and was shared by those two groups of practitioners, whose expertise intersected in the machine and overlapped on the instruction set. Both groups apparently looked upon programming as incidental to their respective concerns. Working with the model of the Turing machine, mathematical logicians concerned themselves with questions of computability considered independently of any particular device, while electrical engineers concentrated on the synthesis and optimization of switching circuits for specific inputs and outputs.[5] Numerical analysts embraced the machine as part of their subject and hence took programming it as part of their task.[6] As the number of computers in use in England in 1953 reached 150, the editor of *Faster than Thought*, B.V. Bowden of Ferranti, Ltd., was unusual in pointing out the difficulties and inefficiencies of programming and in wondering where the programmers would come from.

> We have yet to analyse, for we have almost ignored them, the restrictions on machine performance which are due to the difficulties experienced by the operators who have to prepare programmes for

4. See, for example, Michael R. Williams, *A History of Computing Technology* (Englewood Cliffs, NJ: Prentice-Hall, 1986).

5. M.S. Mahoney, "Computers and mathematics: The search for a discipline of computer science", to appear in the proceedings of the International Symposium on Structures in Mathematical Theories, San Sebastian-Donostia, Spain, September 1990.

6. In a real sense, numerical analysis came into being with the computer. The term itself is of postwar coinage.

them. It is significant that many machines have spent half their working lives in checking programmes and finding mistakes in them and only perhaps a third of the time in straightforward computation. One can deduce from this the startling conclusion that had the machines been a thousand times as fast as they are, their total output would not have been increased by more than about fifty per cent; in the last analysis the correction of programming errors depends almost entirely on the skill and speed of a mathematician, and there is no doubt that it is a very difficult and laborious operation to get a long programme right.[7]

As long as the computer remained essentially a scientific instrument, Bowden's concern found little echo; programming remained relatively unproblematic.

But the computer went commercial in the early '50s. Why and how is another story.[8] With commercialization came rapid strides in hardware — faster processors, larger memories, more efficient peripheral — together with equally rapid expansion of the imaginations of marketing departments. To sell the computer, they spoke not only of high-speed accounting, but of computer-based management. Again, at first few if any seemed concerned about who would write the programs needed to make it useful. IBM, for example, did not recognize "programmer" as a job category nor create a career track for it until the late 1950s.

Companies soon learned that they had reduced the size of their accounting departments only to create ever-growing data processing divisions, or to retain computer service organizations which themselves needed ever more programmers. The process got underway in the late '50s, and by 1968 some 500 companies were producing software. They and businesses dependent on them were employing some 100,000 programmers and advertising the need for 50,000 more. By 1970, the figure stood around 175,000. In this process, programs became "software" in two senses. First, a body of programs took shape (assemblers, monitors, compilers, operating systems, etc.) that transformed the raw machine into a tool for producing useful applications, such as data processing. Second, programs became the objects of production by people who were not scientists, mathematicians, or electrical engineers.

The increasing size of software projects introduced two new elements into programming: separation of design from implementation and management of programmers. The first raised the need for techniques for designing programs —often quite large programs— without writing them and for communicating designs to the programmers. The second raised the need for means of

7. B.V. Bowden (ed.) *Faster That Thought: A Symposium on Digital Computing Machines* (New York, 1953), 96-97.

8. See in particular Bashe, Charles J. et al., *IBM's Early Computers* (Cambridge, MA: MIT Press, 1986) and Kenneth Flamm, *Creating the Computer* (Washington, DC: Brookings Institution, 1988), for American developments and John Hendry, *Innovating for Failure. Government Policy and the Early British Computer Industry* (Cambridge, MA: MIT Press, 1989) for contrasting efforts in Britain.

measuring and controlling the quality of programmers' work.[9] For all the successes of the '60s, practitioners and managers generally agreed that those needs were not being met. Everyone had his favorite horror story. Frederick Brooks published his later as *The Mythical Man-Month* (1975), while C.A.R. Hoare saved his for his Turing Award Lecture in 1980, "The Emperor's Old Clothes".[10] Underlying the anecdotes was a consensus about the nature of the complaints. As F.L. Bauer put it in his report on "Software Engineering" at IFIP 71,

> What have been the complaints? Typically, they were:
> Existing software production is done by amateurs (regardless whether at universities, software houses or manufacturers),
> Existing software development is done by tinkering (at the universities) or by the human wave ("million monkey") approach at the manufacturer's,
> Existing software is unreliable and needs permanent "maintenance, the word maintenance being misused to denote fallacies which are expected from the very beginning by the producer,
> Existing software is messy, lacks transparency, prevents improvement or building on (or at least requires too high a price to be paid for this). Last, but not least, the common complaint is:
> Existing software comes too late and at higher costs than expected, and does not fulfill the promises made for it.
> Certainly, more points could be added to this list.[11]

As an abstract of his paper, Bauer half-jokingly observed that "'Software engineering' seems to be well understood today, if not the subject, then at least the term. As a working definition, it is the part of computer science that is too difficult for the computer scientists." Among the things he seems to have had in mind are precisely the organizational and managerial tasks that one generally associates with engineering rather than science. For he also defined software engineering in all seriousness as "the establishment and use of sound engineering principles to obtain, economically, software that is reliable and works efficiently on real machines", and he proposed to proceed for the moment on the model of industrial engineering. Quite apart from lacking an adequate theory of programming, as forcefully brought out by John McCarthy and others in the early 1960s, computer science (whatever *it* was in the '60s) encompassed nothing akin to project management. Nor did it include the

9. Programming languages were originally aimed at extending access to the computer beyond the professional programmer, who through most of the '60s worked in assembler or machine language. Only in the later '60s, in the course of the developing "crisis" did programming languages take on the role of disciplining programmers, and during most of the '70s unsuccessfully so.
10. *CACM* **24**,2(1981), 75-83; repr. in *BYTE* **6**,10(1981), 414- 425.
11. *Information Processing 71* (Amsterdam: North-Holland Publishing Co, 1972), I, 530-538; at 530.

empirical study of programmers and programming projects to determine the laws by which they behaved.

Traditionally, these had been the concern of engineers, rather than of scientists. That was especially true of American engineers, whose training since the turn of the century had included preparation for managerial responsibilities[12] and who since the turn of the century had been laying claim to superior insight into the organization of efficient production. As Edwin T. Layton showed in *The Revolt of the Engineers*, the claim stemmed from the specific heritage of mechanical engineering and the advances in machine-based industry associated with mass production and the assembly line. One hears beneath much American thinking about software engineering the images and language of the machine shop.

For example, when M.D. McIlroy urged upon fellow participants at Garmisch that they strive toward "Mass-produced software", he was drawing on a repertory of models both for engineering and for management. Seeing software sitting somewhere on the other side of the Industrial Revolution, he proposed to vault it into the modern era.

> We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software.

He left no doubt of whose lead to follow. He continued,

> In the phrase 'mass production techniques', my emphasis is on 'techniques' and not on mass production plain. Of course mass production, in the sense of limitless replication of prototype, is trivial for software. But certain ideas from industrial technique I claim are relevant. The idea of subassemblies carries over directly and is well exploited. The idea of interchangeable parts corresponds roughly to our term 'modularity', and is fitfully respected. The idea of machine tools has an analogue in assembly programs and compilers. Yet this fragile analogy is belied when we seek for analogues of other tangible symbols of mass production. There do not exist manufacturers of standard parts, much less catalogues of standard parts. One may not order parts to individual specifications or size, ruggedness, speed, capacity, precision or character set.

As recent studies of the American machine-tool industry during the 19th and early 20th century have shown, McIlroy could hardly have chosen a more

12. Indeed, at the expense of what Tony Hoare cited as the indispensable common ground of established professional engineering: engineering drawing.

potent model. Between roughly 1820 and 1880, developments in machine-tool technology had increased routine shop precision from .01" to .0001". More importantly, in a process characterized by the economist Nathan Rosenberg as "convergence" machine-tool manufacturers learned how to translate new techniques developed for specific customers into generic tools of their own. So, for example, the need to machine percussion locks led to the development of the vertical turret lathe, which in turn lent itself to the production of screws and small precision parts, which in turn led to the automatic turret lathe.

Moreover, with each advance in precision and automatic operation, machine tools made ever tighter management possible by relocating the machinist's skill into the design of the machine. Increasingly toward the end of the 19th century, workers could be held to close, prefixed standards, because those standards were built into the tools of production. That trend culminated in Ford's machines of production, which automatically produced parts to the tolerances requisite to interchangeability.

As McIlroy knew, mass production was as much a matter of management as of technique. It was not only the sophistication of the individual machines, but also the system by which they were linked and ordered, that transformed American industry. Two figures loomed large in that transformation: Frederick W. Taylor and Henry Ford. The first was associated with "scientific management", the forerunner of modern management science, the latter with the assembly line.

Yet, viewed more closely in light of what was revealed at Garmisch, Taylor's basic principles themselves cast doubt on the applicability of his model to the production of software. The primary obligation of management according to Taylor was to determine the scientific basis of the task to be accomplished. That came down to four main duties:

> *First.* They develop a science for each element of a man's work, which replaces the old rule-of-thumb method.
> *Second.* They scientifically select and then train, teach, and develop the workman, whereas in the past he chose his own work and trained himself as best he could.
> *Third.* They heartily cooperate with the men so as to insure all of the work being done in accordance with the principle of the science which has been developed.
> *Fourth.* There is an almost equal division of the work and the responsibility between the management and the workmen. The management take over all work for which they are better fitted than the workmen, while in the past almost all of the work and the greater part of the responsibility were thrown upon the men.[13]

To what extent computer science could replace rule of thumb in the production of software was precisely the point at issue at the NATO conferences.

13. Frederick Winslow Taylor, *The Principles of Scientific Management* (1911; repr. N.Y.: Norton, 1967), 36-37

Even the optimists agreed that progress had been slow. Unable, then, to fulfil the first duty, programming managers were hardly in a position to carry out the third. Everyone bemoaned the lack of standards for the quality of software. As far as the fourth was concerned, few ventured to say who was best suited to do what in large-scale programming projects.[14]

By 1969 the failure of management to establish standards for the selection and training of programmers was legend. As Dick H. Brandon, the head of one of the more successful software houses, pointed out, the industry at large scarcely agreed on the most general specifications of the programmer's task, managers seeking to hire people without programming experience (as the pressing need for programmers required) had only one quite dubious aptitude test at their disposal, and no one knew for certain how to train those people once they were hired.[15]

Taylor had insisted that productivity was a 50/50 proposition. Management had to play its part through selection, training, and supply of materials and tools. But in the 1960s the science of management (whatever that might be) could not supply what the science of computing (if such there be) had so far failed to establish, namely a *scientific* basis of software production.[16]

What could not be organized by Taylor's methods *a fortiori* lay beyond the

14. In *The Mythical Man-Month: Essays in Software Engineering* (Reading, MA, 1975), Frederick P. Brooks, Jr., manager of IBM's OS/360 project, recounted his own failures in this regard: "It is a very humbling experience to make a multimillion-dollar mistake, but it is also very memorable. I vividly recall the night we decided how to organize the actual writing of external specifications for OS/360. The manager of architecture, the manager of control program implementation, and I were threshing [!] out the plan, schedule, and division of responsibilities. The architecture manager had 10 good men. He asserted that they could write the specifications and do it right. It would take ten months, three more than the schedule allowed.

"The control program manager had 150 men. He asserted that they could prepare the specifications, with the architecture team coordinating; it would be well done and practical, and he could do it on schedule. Furthermore, if the architecture team did it, his 150 men would sit twiddling their thumbs for ten months.

"To this the architecture manager responded that if I gave the control program team the responsibility, the result would *not* in fact be on time, but would also be three months late, and of much lower quantity. I did, and it was. He was right on both counts. Moreover, the lack of conceptual integrity made the system far more costly to build and change, and I would estimate that it added a year to debugging time". (47-48)

15. Dick H. Brandon, "The Economics of Computer Programming", in George F. Weinwurm (ed.), *On the Management of Computer Programming* (Princeton: Auerbach, 1970), Chap.1. Brandon evidently viewed management through Taylorist eyes, but he was clear-sighted enough to see that computer programming failed to meet the prerequisites for scientific management. For an analysis of why testing was so unreliable, see R.N. Reinstedt, "Results of a Programmer Performance Prediction Study", *IEEE Trans. Engineering Management* (12/67), 183-87, and Gerald M. Weinberg's *The Psychology of Computer Programming* (NY, 1971), Chap.9.

16. Taylor had also laid particular emphasis on wage structures that encourage full production. The essence of his "differential piece rate" offered the worker a choice to produce at the optimal rate or not; it was a choice about the pace at which to work, Taylor's or the worker's. Brandon pointed out that the anarchic nature of programming meant that management had to depend on the workers to determine the pace of a project and that the insatiable market for programmers meant that management had little control at all over the wage structure.

reach of the other major American model of productivity: Ford's assembly line. The essential feature of Ford's methods is the relocation of skill from the worker to the machines of production, which the workers merely attended. Ford's worker had no control over the quality or the quantity of the work he produced. The machines of production determined both. To Fordize software production would mean providing the computer analogue of automatic machinery. It would therefore mean that one would have to design software systems that generate software to meet set standards of reliability. When McIlroy spoke of "mass-produced software", he was speaking the language of Ford's model.

As noted earlier, quality control was another feature of Ford's system. His machines guaranteed accuracy to within 0.0001" for complete interchangeability of parts. The analogue in computing that McIlroy was espousing thus brings the question of production back around to the question of the nature of computer science. Some of those at NATO thought of software design in terms of experimentation: write the program, put it on the computer, and start debugging (as Ford put it, "Let's turn it over and see why it won't start".). Programming then looks like experimental science or engineering in viewing the reliability of software in terms of tolerances. But Edsger W. Dijkstra vehemently advocated a quite different notion. "Program testing can be used to reveal the presence of errors", he argued, "but never to show their absence"! (or even, he might have added, convergence on their absence).[17] Programming should aspire to mathematics, that is, it should seek means of verifying the correctness of programs mathematically. Rather than accepting the inevitability of bugs and devising elaborate tests to find them, one should prove that programs will work properly. Dijkstra was calling the means of such proof "structured programming" and was seeking to build its structure into programming languages and their compilers. It was the mathematically oriented computer scientists who seem to have been responding to questions of productivity with an eye toward Ford's methods.

A survey of the software engineering literature of the past decade reveals three salient features of the field. First, the problems that spawned software engineering remain largely unresolved. Ten years after the NATO Conference, R.W. Floyd in his Turing Award Lecture of 1978 quoted Robert Balzer to the effect that

17. In "Structured Programming", his contribution to the Rome NATO conference, 1969; *NRB*, 223.

It is well known that software is in a depressed state. It is unreliable, delivered late, unresponsive to change, inefficient, and expensive. Furthermore, since it is currently labor intensive, the situation will further deteriorate as demand increases and labor costs rise.[18]

To this Floyd could only add:

> If this sounds like the famous 'software crisis' of a decade or so ago, the fact that we have been in the same state for ten or fifteen years suggests that 'software depression' is a more apt term.

One does not have to look hard or far for similar expressions in the current software literature.

Indeed, it suffices to read any issue of ACM's Software Engineering Notes, with its list of new items describing the failures of software systems, including the lost of life, limb, property, money and time suffered by users of improperly functioning programs. The disclaimers of liability that routinely accompany software bear witness to how far software engineering lies from the "established branches of engineering". Thus, despite the hallmarks of an established discipline — societies, journals, curricula, research institutions — software engineering remains a relatively soft concept. The definition has not changed much; a recent version speaks of "the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software".[19] The meaning of the definition depends on how its central terms are interpreted, in particular what principles and methods from engineering, science, and mathematics are applied to software.

Hence, behind the discussions of software engineering lie models drawn from other realms of engineering, often without much reflection on their applicability. In most cases, they are the same models that informed people's thinking when software engineering was a new idea. For example, McIlroy's notion of assembling components became modular programming and then object-oriented programming, but the underlying model remained that of production by interchangeable parts. Automatic programming has shifted meaning as its scope has broadened from compiling to implementation of design, but it continues to rest on the model of automatic machine tools. Software engineering has not progressed far beyond its roots. Perhaps its roots are the reason.

18. Robert Balzer, "Imprecise Program Specification", *Report ISI/RR-75-36*, Information Science Institute, Dec.1975; quoted by Robert W. Floyd, "The Paradigms of Programming", *CACM* **22**,8(1978), 455-460; at 456.

19. Watts Humphrey, "The Software Engineering Process: Definition and Scope", in *Representing and Enacting the Software Process: Proceedings of the 4th International Software Process Workshop* (New York: ACM Press, 1989), 82.